Available online at www.sciencedirect.com

SciVerse ScienceDirect

Physics Procedia 37 (2012) 1965 – 1972

Physics Procedia

TIPP 2011 – Technology and Instrumentation in Particle Physics 2011

Performance Study of GPUs in Real-Time Trigger Applications for HEP Experiments

W. Ketchum[a], S. Amerio[b], D. Bastieri[b,c], M. Bauce[b,c], P. Catastini[d], S. Gelain[c], K. Hahn[d], Y. K. Kim[d,a], T. Liu[d], D. Lucchesi[b,c], G. Urso[e]

[a] *University of Chicago*
[b] *INFN Padova*
[c] *University of Padova*
[d] *FNAL*
[e] *ORMA Software*

Abstract

Graphical Processing Units (GPUs) have evolved into highly parallel, multi-threaded, multicore powerful processors with high memory bandwidth. GPUs are used in a variety of intensive computing applications. The combination of highly parallel architecture and high memory bandwidth makes GPUs a potentially promising technology for effective real-time processing for High Energy Physics (HEP) experiments. However, not much is known of their performance in real-time applications that require low latency, such as the trigger for HEP experiments. We describe an R&D project with the goal to study the performance of GPU technology for possible low latency applications, performing basic operations as well as some more advanced HEP lower-level trigger algorithms (such as fast tracking or jet finding). We present some preliminary results on timing measurements, comparing the performance of a CPU versus a GPU with NVIDIA's CUDA general-purpose parallel computing architecture, carried out at CDF's Level-2 trigger test stand. These studies will provide performance benchmarks for future studies to investigate the potential and limitations of GPUs for real-time applications in HEP experiments.

Keywords: HEP Trigger, GPU, CPU, Fast track-fitting, Tracking trigger

1. Introduction

Commercially available graphical processing units (GPUs) have increased greatly in their performance capabilities over the past decade, outpacing improvements in the number of floating-point calculations per second and overall memory bandwidth in traditional central processing units (CPUs) [1]. Driven by the high demand of graphics-intense applications in PCs, GPUs have evolved into powerful multicore processors, specializing in highly parallelized, multi-threaded computations. This is accomplished by devoting a greater number of resources to data-processing, at the expense of quick-access memory and simplicity of flow controls. Additionally, NVIDIA has developed a general-purpose parallel computing architecture, CUDA, with a new parallel programming model and instruction set [1]. CUDA contains a software environment

 doi:10.1016/j.phpro.2012.04.108

that allows developers to use C/C++ as a high-level programming language for the GPU, making it more accessible to the general user.

Typically, low-level trigger systems of high energy physics (HEP) experiments have used dedicated hardware and/or PCs with CPUs running decision algorithms. For example, the CDF Level 2 (L2) trigger system used dedicated hardware combined with a single PC running an optimized Linux kernel to perform real-time trigger decision algorithms, and achieved a system-level latency on the order of tens of microseconds [2]. The serial nature of running algorithms on a CPU, however, limits the performance of these systems and makes it harder to scale for experiments that face much higher occupancies, like those that are expected at the LHC due to the greater number of multiple interactions per proton bunch crossing, and a smaller time between bunch crossings. A processing device with a large number of parallel processing threads available, like a GPU, may be able to better address this issue of scaling. However, little is known about GPU performance, both in terms of HEP trigger algorithm speed and latency overheads, in low-latency environments ($\sim 100\,\mu$s).

We present initial studies on measuring the timing performance of a GPU using a simplified calculation mimicking fast track-fitting, like that performed in the hardware-based Silicon Vertex Trigger (SVT) system at CDF [3, 4, 5]. To provide a basis for comparison, we also study the performance of a CPU doing the same calculation. Our setup, described in Sec. 2, is unique in that one measures the latency using hardware independent of the PC, and without relying on internal software time stamps.

2. Setup

The test setup, shown in Fig. 1, is based on the CDF L2 trigger test stand, consisting of a VME crate running as one partition of the CDF DAQ system, and two general-purpose PULSAR (PULser And Recorder) boards [6]. One PULSAR board is configured as an S-LINK [7] transmitter (Tx) that, upon a user-generated L1 accept, sends user-defined patterns which mimic raw silicon hits data. The transmitter simultaneously sends two copies of these patterns: one copy is sent to a PC, while the other is sent to another PULSAR board, configured as an S-LINK receiver (Rx). The receiver records the time of arrival of the S-LINK packet coming directly from the Tx (t_1), and of the S-LINK packet containing results from the PC (t_2) with respect to the L1 accept. The total latency is then $t_2 - t_1$.

The PC has an Intel Core i7-930 CPU and is instrumented with an nVidia GeForce GTX 285 GPU on a PCIe slot. A comparison of some of the specifications of these devices is shown in Tab. 1. Additionally, the PC is equipped with two S-LINK-to-PCI interface cards on its PCI-X slots: the CERN FILAR (Four Input Links for Atlas Readout) [8] receives S-LINK packets from the Tx, and an S32PCI64 [9] SOLAR (Single Output Link for Atlas Readout) sends algorithm results in S-LINK packets from the PC to the Rx.

	Intel Core i7-930 CPU	nVidia GeForce GTX 285 GPU
Microprocessors	1	30
Cores	4	240
Threads (per microprocessor)	8	1024
Cache Size (per microprocessor)	8 MB	8 kB

Table 1. Comparison of the CPU and GPU used in these studies.

3. Benchmark Algorithm

Many tasks performed by trigger systems may benefit from the parallelization available in a GPU (*e.g.*, jet clustering and track finding). For this study, we choose as a benchmark a simplified fast track-fitting algorithm which was used in CDF's Silicon Vertex Trigger (SVT) [3, 4]. This algorithm uses a linearized

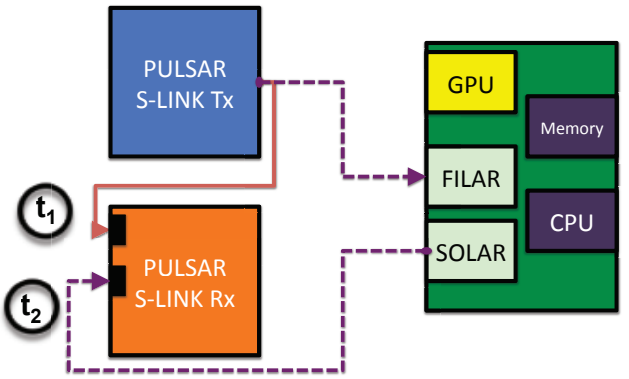

Fig. 1. Schematic of the test stand setup for latency measurements of our track-fitting procedure. We use two PULSARs, one transmitter and one receiver, and send data from the transmitter to the PC's FILAR, and back from the PC to the receiver via a SOLAR. The receiver measures the arrival time of data directly from the transmitter (t_1) and the PC (t_2) since a user-generated L1 accept.

approximation to track-fitting as implemented in hardware (described in greater detail in [5]). With the SVT approach, the determination of the track parameters (p_i) is reduced to a simple scalar product:

$$p_i = \vec{f_i} \cdot \vec{x_i} + q_i$$

where $\vec{x_i}$ are input silicon hits, and $\vec{f_i}$ and q_i are pre-defined constant sets. For each set of hits, the algorithm computes the impact parameter d_0, the azimuthal angle ϕ, the transverse momentum p_T, and the χ^2 of the fitted track by using simple operations such as memory lookup and integer addition and multiplication. In our testing of the track-fitting algorithm, each S-LINK word in the user-defined test pattern is treated to represent a set of silicon hits. While the track-fitting algorithm itself is simple, it must be performed on many combinations of hits in a given event, especially in high-occupancy environments. It is an ideal benchmark for testing performance of a GPU, using massive parallelization, at low latencies.

4. Measurements

The data flow through the test setup can be divided into the following steps:

1. S-LINK packets from the Tx, containing the input silicon hits information, are received by the FILAR and put into the CPU's memory via direct memory access;
2. if the track-fitting algorithm is to be performed in the GPU, these hits are copied from the CPU (also referred to as the "host") to the memory of the GPU (also referred to as the "device");
3. the benchmark algorithm is performed, either in the CPU or the GPU;
4. if the algorithm was performed in the GPU, the results are copied from the GPU back to the CPU;
5. the results are sent in S-LINK packets from the SOLAR to the Rx.

This study focuses on the measurement of the contribution of each of these steps to the total latency. While we perform measurements for a variety of number of S-LINK words (each 32-bit), by default we send 500 words as input, perform the track-fitting algorithm on all 500 input words, and store the output in 2000 words (four output words for each input word, as the fit returns four track parameters for each set of input hits). After sending the output to the Rx, we perform a check using the CPU to ensure all calculations were done properly in the GPU, but this latency is not included in the measurement. For simplicity, we run at a sufficiently low rate to ensure all calculations are finished before the next test pattern is sent to the PC.

4.1. *Tx ⟶ PC ⟶ Rx Data Transfer Latency*

We first measure the latency attributable to the data transfer between the Tx/Rx and the PC (items 1 and 5 above). This latency, t_{IO}, is an overhead due only to data transfer. For these measurements, shown in Fig. 2 and summarized in Tab. 2, we send 1, 10, 100 or 500 words from the Tx, and send an acknowledgement from the PC to the Rx via the SOLAR after all input words are received. There is an overhead, independent of the number of words being sent, of about 6 μs. The overall latency increases as a function of the input words. The latency for 500 input words, our default, is $t_{IO} \sim 21$ μs.

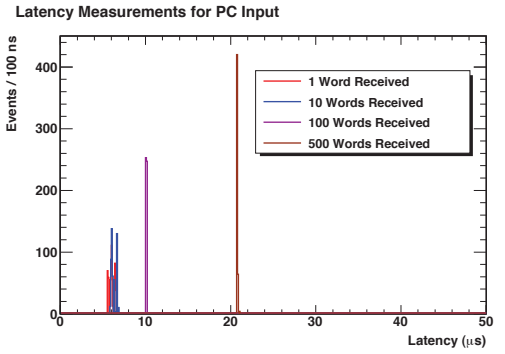

Fig. 2. Latency for transferring data from PULSAR Tx into the PC and acknowledging acceptance.

N_{words} Input	Mean Latency (μs)
1	6
10	6
100	10
500	***21***

Table 2. The mean of the latency for transferring data from PULSAR Tx into the PC and acknowledging acceptance. The number of input words used in our default configuration is shown in bold.

4.2. *CPU (Host) ⟷ GPU (Device) Data Transfer Latency*

In this section, we present measurements of the latency for copying data from host to device and copying results from the device to the host without performing our benchmark algorithm (steps 2 and 4 in the list above). The transfer of data between the CPU and GPU is an additional overhead on the latency for GPU applications that must be considered. These measurements are shown in Fig. 3 and summarized in Tab. 3. In all cases we send 500 words from the Tx, and send acknowledgement to the Rx upon completion of the data transfer, thus the t_{IO} is included in each of the latency measurements.

We find that when copying data from the CPU to GPU, there is an average latency of 26 μs for copying 500 words, and it does not depend much on the number of words copied. Transferring data in the reverse direction, from the GPU to the CPU, takes considerably longer, highlighting an asymmetry in the memory transfer between host and device. For the default configuration, copying 2000 words from the GPU to the CPU, the average latency is 39 μs. We also see a larger spread in the latency times than we did in t_{IO}, indicating some jitter in the data transfer process. After removing the PULSAR↔PC data transfer overhead (t_{IO}, see Fig. 2), the latency for copying from host to device is $t_{H\to D} \sim 5$ μs, and for device to host is $t_{D\to H} \sim 18$ μs in the default configuration. In reality, the number of output words sent back to the CPU can be greatly reduced for in an actual track-fitting implementation as most hits combinations will have poor fits with large χ^2 values, and could be rejected in the GPU.

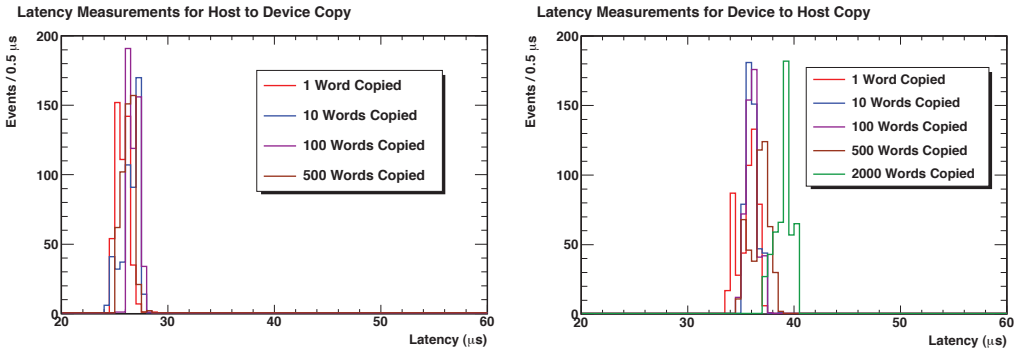

Fig. 3. Latency for transfering data from CPU to GPU (*left*) and from GPU to CPU(*right*).

N_{words} Copied	Mean Latency (μs)	Latency − t_{IO} (μs)
(CPU→GPU)		$t_{H\to D}$
1	26	5
10	26	6
100	27	6
500	***26***	***5***
(GPU→CPU)		$t_{D\to H}$
1	36	15
10	36	15
100	36	15
500	37	16
2000	***39***	***18***

Table 3. The mean latency for transfering data from CPU to GPU (*left*) and from GPU to CPU(*right*). The default configuration latencies are shown in bold.

4.3. Latency of Benchmark Algorithm

With the latency overhead due to data-flow characterized, we then measure the total latency when performing the benchmark algorithm described in Sec. 3. We compare the latency for running the track-fitting algorithm in the CPU in series, where the fits are handled one at a time, and in the GPU, where the fits are performed in parallel: each computational thread in the GPU performs a single fit for a set of hits. For the cases where $N_{fits} \leq 100$, we assign all threads to one block in the GPU. When we run calculations performing fits for all 500 input words, we use a total of 5 blocks. Also, when performing measurements for the GPU, we copy to the GPU all 500 input words, and copy back 2000 output words, regardless of the number of calculations performed.

The latency measurements, including running the track-fitting algorithm, are shown in Fig. 4, and summarized in Tab. 4. We see a natural increase in the latency as we increase the number of fits performed in the CPU, with a total calculation time of about 13 ns/fit. For the GPU, the spread in times are much larger, but are less dependent on the increase in the number of fits. However, the overall time for performing the calculations in the GPU is longer—about 40 μs—than the time taken in the CPU. Contributions from the memory copy operations are a significant part of this difference: from Sec. 4.2, the overhead time for transferring data to and from the GPU is $t_{H\to D} + t_{D\to H} \sim 24$ μs.

4.4. Latency Dependence on Memory Types Used

Latency from GPU calculations also depends on the type of memory used inside the GPU. GPUs that follow the CUDA architecture have a variety of types of device memory [1]. The *global memory* has the

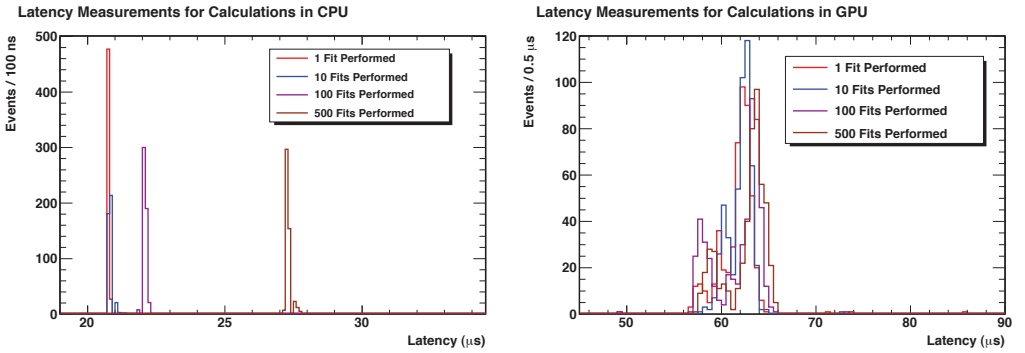

Fig. 4. Latency when performing our track-fitting algorithm in the CPU (*left*) and GPU (*right*), varying the number of fits performed.

N_{fits}	Mean Latency, CPU (μs)	Mean Latency, GPU (μs)
1	21	62
10	21	62
100	22	62
500	27	63

Table 4. The mean of total latency when performing our track-fitting algorithm in the CPU and the GPU, varying the number of fits performed. Further detail is provided in Fig. 4.

largest size, is accessible to all threads in the GPU calculations, and has both read and write capabilities; however, it is not located on the actual microprocessor chip, and access to it tends to be slow. *Constant memory* is also located off of the chip, but it can be cached, allowing quicker access. Constant memory is, however, read-only, and limited in size. *Register memory* is the fast memory available directly on the chip used by the thread, but its size is severely limited, and not directly accessible from the host (CPU). It is important to study the latency for using different kinds of GPU memory in order to optimize performance for a given application.

In the previous measurements for the GPU, the constant sets were stored in the constant memory. In Fig. 5 we show the latency for performing the track-fitting algorithm in the GPU when storing the constant sets in global memory, constant memory, and the register memory. We find differences of up to 7 μs for different memory usage. These measurements show that not only is optimizing memory transfer between the CPU and GPU important, but that the memory management within the GPU has a significant effect on the overall latency.

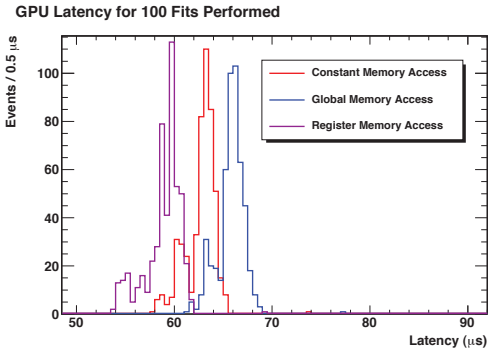

Fig. 5. Latency when performing our track-fitting algorithm in the GPU, performing 100 fits, varying where in the GPU memory the constant sets are stored.

Memory Location	Mean Latency (μs)
Global	66
Constant	63
Register	59

Table 5. The mean of latency measurements of performing our track-fitting algorithm in the GPU for various locations of the constant sets used in the calculations. See Fig. 5 for more detail.

5. Summary and Outlook

We have presented timing measurements of the performance of a simplified track-fitting algorithm in a GPU to determine the potential for using GPUs in low-level HEP trigger systems. These measurements have been performed at the CDF L2 trigger test stand, where latencies can be measured without using software time stamps in the PC. The total latency for our benchmark algorithm in the GPU is < 100 μs, showing promise for future use in HEP triggers. We have also compared performance for a GPU to that for a CPU.

Our studies indicate some properties of GPUs that will make their application in HEP trigger systems particularly challenging. For instance, the latency due to the transfer of data from the CPU to the GPU and vice versa can be significant. This data transfer overhead makes the CPU better-suited for our simplified track-fitting algorithm performing a small number of fits, as was shown in Sec. 4.4. However, the CPU latency increases with the number of fits performed, while performance in the GPU remains relatively constant with the number of fits. We expect that as the complexity of the algorithm and number of fits increases, the GPU's advantages over the CPU will become more pronounced. We plan to conduct further studies to investigate this.

Additional strategies to reduce the CPU↔GPU data transfer latency exist [10]. A limited amount of data transfers can be done using "pinned" memory, which may provide higher data transfer rates than "pageable" memory. Recent GPUs also have a "zero-copy" feature that allows the GPU threads to directly access some host memory. Another new feature from nVidia, DirectGPU, allows the GPU to share some pinned memory directly with other devices. Studies using these strategies to reduce latency are underway.

Another factor in optimizing performance in the GPU involves memory access and allocation within the GPU. Memory management within the GPU will, by nature, be application specific, and we are considering various strategies for keeping the latency due to internal memory access low as we increase the complexity of the calculations we perform in the GPU.

6. Acknowedgements

We would like to thank Simon Kwan of Fermilab for his early support of this project. Silvia Amerio was supported by a Marie Curie International Outgoing Fellowship within the 7th European Community Framework Programme.

References

[1] NVIDIA, NVIDIA CUDA C Programming Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf (2011).
[2] K. Anikeev, et al., CDF level 2 trigger upgrade, IEEE Trans.Nucl.Sci. 53 (2006) 653–658. doi:10.1109/TNS.2006.871782.
[3] B. Ashmanskas, et al., The CDF silicon vertex trigger, Nucl.Instrum.Meth. A518 (2004) 532–536. arXiv:physics/0306169.
[4] J. A. Adelman, et al., The Silicon Vertex Trigger upgrade at CDF, Nucl.Instrum.Meth. A572 (2007) 361–364. doi:10.1016/j.nima.2006.10.383.
[5] S. Amerio, et al., The GigaFitter: Performance at CDF and perspectives for future applications, J.Phys.Conf.Ser. 219 (2010) 022001.
[6] For more information, see http://hep.uchicago.edu/~thliu/projects/Pulsar/.
[7] E. van der Bij, et al., S-LINK, a data link interface specification for the LHC era, IEEE Trans.Nucl.Sci. 44 (1997) 398–402, http://hsi.web.cern.ch/HSI/s-link/. doi:10.1109/23.603679.

[8] W. Iwanski, et al., http://hsi.web.cern.ch/HSI/s-link/devices/filar/Welcome.html.
[9] W. Iwanski, et al., http://hsi.web.cern.ch/HSI/s-link/devices/s32pci64/.
[10] NVIDIA, NVIDIA CUDA C Best Practices Guide, http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Best_Practices_Guide.pdf (2011).